
Commodore Assembler Development Package

User's Manual

A discussion and description
of the Assembly Language
and assembly process for
MCS-650X Microprocessors

Appropriate for use with:

- Series 2001 Computers
- Series 3000 Computers
- Series 8000 Computers
- Model 2040 Floppy
- Model 8050 Floppy

Part Number 321601



© 1980 Commodore Business Machines, Inc.

The Commodore Assembler Development Package
is comprised of:

- User's Manual (Part Number 321601)
- Software Program (Part Number 321511)
- MCS6500 Instruction Set Summary
- MCS6500 Microcomputer Family
Hardware Manual (6500-10A)
- MCS6500 Microcomputer Family
Programming Manual (6500-50A)
- MCS6522 Versatile Interface Adapter
(Preliminary Data Sheet)

Table of Contents

Chapter 1	Introduction	1
Chapter 2	Instruction Format	3
	Symbolic	7
	Constants	7
	Relative	8
	Implied	8
	Indexed Indirect	9
	Indirect Indexed	10
Chapter 3	Assembler Directives	11
Chapter 4	Output Files	15
	Listing File	15
	Interface File	15
	Error List	16
	**A,X,Y,S,P RESERVED	17
	**.A MODE NOT ALLOWED	17
	**INVALID ADDRESS	17
	**FORWARD REFERENCE	17
	**ILLEGAL OPERAND TYPE	18
	**IMPROPER OP CODE	19
	**CAN'T EVAL EXPRESSION	19
	**INDEX MUST BE X OR Y	19
	**LABEL START NEED A-Z	20
	**LABEL TOO LONG	20
	**NON-ALPHANUMERIC	20
	**DUPLICATE SYMBOL	21
	**INDIRECT OUT OF RANGE	21
	**PC NEGATIVE--RESET O	21
	**RAN OFF END OF CARD	21
	**BRANCH OUT OF RANGE	22
	**UNDEFINED DIRECTIVE	22
	**UNDEFINED SYMBOL	22
Appendix I	Universal DOS Support	25
Appendix II	Instructions For The Mini-Editor	27
	Loading The Mini-Editor	27
	Operation of The Mini-Editor	28
	Mini-Editor Commands	28
	Auto Line Numbering	28
	Break Command	28
	Change String	28
	Delete	29
	Find String	30
	Formatted Print	30
	GET Files	30
	KILL Command	30
	Resequene Lines	31
	PUT Command	
Appendix III	Operation of the PET Assembler	33
Appendix IV	PET Loaders	35
Appendix V	2001 Series I/O Map	37

Chapter

1

INTRODUCTION

This manual describes the Assembly Language and assembly process for programs for the MCS-650X series of microprocessors. Several assemblers are available for program development, and while they are all slightly different in detail of use, they are the same in substance.

The process of translating a mnemonic or symbolic form of a computer program to actual machine code is called assembly, and a program which performs the translation is an assembler. The symbols used and rules of association for those symbols are the Assembly Language. In general, one Assembly Language statement will translate into one machine instruction. This distinguishes an assembler from a compiler which may produce many machine instructions from a single statement. An assembler which executes on a computer other than the one for which code is generated is called a cross-assembler. Use of cross-assemblers for program development for microprocessors is common since often a microcomputer system has fewer resources than are needed for an assembler. However, in the case of the Commodore system this is not true. With a floppy disk and printer the system is well suited for software development.

Normally digital computers use the binary number system for representation of data and instructions. Computers understand only ones and zeroes corresponding to an "on" or "off" state. Users, on the other hand, find it difficult to work with the binary number system and hence use a more convenient representation such as octal (base 8), decimal (base 10) or hexadecimal (base 16). Two representations of the MCS-650X operation to "load" information into an "accumulator" are:

```
10101001 (binary)
A9       (hexadecimal)
```

An instruction to move the value 21 (decimal) to the accumulator is:

```
A9 15      (hexadecimal)
```

Users still find numeric representations of instructions tedious to work with, and hence, have developed symbolic representations. For example, the preceding instruction might be written as:

```
LDA    #21
```

In this case, LDA is the symbol for A9, Load the Accumulator. A computer program used to translate the symbolic form LDA to numeric form A9 is called an assembler. The symbolic program is referred to as source code and the numeric program is the object code. Only object code can be executed on the processor.

Each machine instruction to be executed has a symbolic name referred to as an operation code (OPCODE). The OPCODE for "store accumulator" is STA. The OPCODE for "transfer accumulator to index X" is TAX. The 56 OPCODES for MCS-650X processors are detailed in Chapter 2. A machine instruction in Assembly Language consists of an OPCODE and perhaps OPERANDS, which specify the data on which the operation is to be performed.

Instructions may be labelled for reference by other instructions as shown in:

```
L2    LDA    #12
```

The label is L2, the OPCODE is LDA, and the OPERAND is #12. At least one blank must separate the three parts (fields) of the instruction. Additional blanks are inserted by the assembler for ease of reading. Instructions for the MCS-650X processors have at most one OPERAND and many have none. In these cases the operation to be performed is totally specified by the OPCODE as in CLC (Clear the Carry Bit).

Programming in Assembly Language requires learning the instruction set (OPCODES), addressing conventions for referencing data, the data structures within the processor, as well as the structure of Assembly Language programs. The user will be aided in this by reading and studying the MCS-650X hardware and programming manuals supplied with this development package.

Chapter 2

INSTRUCTION FORMAT

Assembler instructions for the MCS-650X assembler are of two basic types according to function:

- Machine instructions, and
- Assembler directives

Machine instructions correspond to the 56 operations implemented on the MCS-650X processors. The instruction format is:

(label) (OPCODE) (OPERAND) (comments)

Fields are bracketed to indicate that they are optional. Labels and comments are always optional and many OPCODES such as RTS (Return from Subroutine) do not require OPERANDS. A typical instruction showing all four fields is:

LOOP LDA BETA,X FETCH BETA INDEXED BY X

A field is defined as a string of characters separated by a blank space or tab characters. The list of OPCODES for the MCS-650X processors is shown in Table 1.

A label is an alphanumeric string of from one to six characters, the first of which must be alpha. A label may not be any of the 56 OPCODES, nor any of the special single characters, i.e. A, S, P, X or Y. These special characters are used by the assembler to reference the:

- Accumulator (A)
- Stack pointer (S)
- Processor status (P)
- Index registers (X and Y)

A label may begin in any column provided it is the first field of an instruction. Labels are used on instructions as branch targets and on data elements for reference in OPERANDS.

Table 1. MCS-650X Microprocessor Instruction Set - OPCODES

ADC	Add with Carry to Accumulator	LDX	Transfer Memory to Index X
AND	"AND" to Accumulator	LDY	Transfer Memory to Index Y
ASL	Shift Left One Bit (Memory or Accumulator)	LSR	Shift One Bit Right (Memory or Accumulator)
BCC	Branch on Carry Clear	NOP	Do Nothing - No Operation
BCS	Branch on Carry Set	ORA	"OR" Memory with Accumulator
BEQ	Branch on Zero Result	PHA	Push Accumulator on stack
Bit	Test Bits in Memory with Accumulator	PHP	Push Processor Status on Stack
BMI	Branch on Result Minus	PLA	Pull Accumulator from Stack
BMI	Branch on Result Minus	PLP	Pull Processor Status from Stack
BNE	Branch on Result not Zero	ROL	Rotate One Bit Left (Memory or Accumulator)
BNE	Branch on Result not Zero	ROR	Rotate One Bit Right (Memory or Accumulator)
BPL	Branch on Result Plus	RTI	Return From Interrupt
BRK	Force an Interrupt or Break	RTS	Return From Subroutine
BVC	Branch on Overflow Clear	SBC	Subtract Memory and Carry from Accumulator
BVS	Branch on Overflow Set	SEC	Set Carry Flag
CLC	Clear Carry Flag	SED	Set Decimal Mode
CLD	Clear Decimal Mode	SEI	Set Interrupt Disable Status
CLI	Clear Interrupt Disable Bit	STA	Store Accumulator in Memory
CLV	Clear Overflow Flag	STX	Store Index X in Memory
CMP	Compare Memory and Accumulator	STY	Store Index Y in Memory
CPX	Compare Memory and Index X	TAX	Transfer Accumulator to Index X
CPY	Compare Memory and Index Y	TAY	Transfer Accumulator to Index Y
DEC	Decrement Memory by One	TSX	Transfer Stack to Index X
DEX	Decrement Index X by One	TXA	Transfer Index X to Accumulator
DEY	Decrement Index Y by One	TXS	Transfer Index X to Stack Register
EOR	Exclusive-OR Memory with Accumulator	TYA	Transfer Index Y to Accumulator
INC	Increment Memory by One		
INX	Increment X by One		
INY	Increment Y by One		
JMP	Jump to new Location		
JSR	Jump to New Location Saving Return Address		
LDA	Transfer Memory to Accumulator		

The OPERAND portion of an instruction specifies either an address or a value. An address may be computed by expression evaluation and the assembler allows considerable flexibility in expression formation. An Assembly Language expression consists of a string of names and constants separated by operators +, -, *, and / (add, subtract, multiply, and divide). Expressions are evaluated by the assembler to compute OPERAND addresses. Expressions are evaluated left to right with no operator precedence and no parenthetical grouping. Note that expressions are evaluated at assembly time and not execution time.

Any string of characters following the OPERAND field is considered a comment and is listed, but not further processed. If the first non-blank character of any record is a semi-colon (;), the record is processed as a comment. On instructions which require no OPERAND, comments may follow the OPCODE. At least one separating character (space or horizontal tab) must separate the fields of an instruction.

There are ten assembler directives used to reserve storage and direct information to the assembler. Nine have symbolic names with a period as the first character. The tenth, a symbolic equate, uses an equals sign (=) to establish a value for a symbol. A list of the directives are given below and their use is explained in a later section.

.BYTE	.WORD	.DBYTE	.PAGE	.SKIP			
.OPT	.END	.FILE	.LIB	=	or	*	=

Labels and symbols other than directives may not begin with a period.

A typical MCS-650X assembler program segment is shown in Table 2. This table is presented primarily to show the form of the information output by the assembler.

Table 2. Segment of an MCS-650X Program

LINE#	LOC	CODE	LINE	
0054	0295			; LINE OF BASIC TEXT TO ALLOW
0055	0295			; USER TO TYPE 'RUN'
0056	0295			* =\$400
0057	0400	00		. BYT 0, 13, 4, 10, 0, 158 ; 10 SYS
0057	0401	0D		
0057	0402	04		
0057	0403	0A		
0057	0404	00		
0057	0405	9E		
0058	0406	28 31		. BYT '(1039)', 0, 0, 0
0058	040C	00		
0058	040D	00		
0058	040E	00		
0061	040F	A9 00	LOAD	LDA #0
0062	0411	8D 83 02		STA CHAN
0063	0414	8D 94 02		STA OBJLEN
0064	0417	8D 7F 02		STA RECCNT
0065	041A	8D 80 02		STA RECCNT+1
0066	041D	8D 7C 02		STA OFFSET ; ZERO OFFSET
0067	0420	8D 7D 02		STA OFFSET+1
0068	0423	A2 40		LDX #OFFMSG-MSGS ; ASK FOR OFFSET
0069	0425	20 89 05		JSR MSG
0070	0428	20 E1 F1	LD005	JSR BASIN ; GET A CHARACTER
0071	042B	C9 20		CMP ##20 ; SPACE IS ERROR
0072	042D	F0 F9		BEQ LD005
0073	042F	C9 0D		CMP ##0D ; RETURN IS OK (DEFAULT OFFSET TO
0074	0431	F0 10		BEQ LD010
0076	0433	20 BE E7		JSR RDOB1 ; GET THE HIGH BYTE
0077	0436	90 D7		BCC LOAD ; ERROR START OVER...
0078	0438	8D 7D 02		STA OFFSET+1
0079	043B	20 B6 E7		JSR RDOB ; GET THE LOW BYTE
0080	043E	90 CF		BCC LOAD ; ERROR START OVER....
0081	0440	8D 7C 02		STA OFFSET
0082	0443		LD010	
0084	0443	A2 8E		LDX #OBJMSG-MSGS
0085	0445	20 89 05		JSR MSG
0086	0448	A2 28		LDX #40
0087	044A	8E 81 02		STX INDEX
0088	044D	CE 81 02	LD10	DEC INDEX
0089	0450	F0 BD		BEQ LOAD
0090	0452	20 E1 F1		JSR BASIN
0091	0455	C9 20		CMP ##20
0092	0457	F0 F4		BEQ LD10
0093	0459	C9 0D		CMP ##D
0094	045B	D0 03		BNE GOLOAD
0095	045D	4C 79 05		JMP DONE ; RETURN EXITS LOADER...
0096	0460	A2 00	GOLOAD	LDX #0
0097	0462	8E 94 02		STX OBJLEN

SYMBOLIC

Perhaps the most common OPERAND addressing mode is the symbolic form as in:

```
LDA    BETA    PUT BETA VALUE IN ACCUMULATOR
```

In the example BETA references a byte in memory that is to be loaded into the accumulator. BETA is an address at which the value is located. Similarly, in the instruction

```
LDA    ALPHA+BETA
```

the address ALPHA+BETA is computed by the assembler and the value at the computed address is loaded into the accumulator.

Memory associated with the MCS-650X processors is segmented into pages of 256 bytes each. The first page, page zero, is treated differently by the assembler and by the processor for optimization of memory storage space. Many of the instructions have alternate operation codes if the OPERAND address is in page zero memory. In those cases the address requires only one byte rather than the normal two. For example, if BETA is located at the byte 4B in page zero memory, then the code generated for

```
LDA    BETA
```

is A5 B4. This is called page zero addressing. If BETA is at 01 3C in memory page one the code generated is AD 3C 10. This is an example of absolute addressing. Thus, to optimize storage and execution time, a programmer should design with data areas in page zero memory whenever possible. Note that the assembler makes decisions on which form to use based on OPERAND address computation.

CONSTANTS

Constant values in Assembly Language can take several forms. If a constant is other than decimal a prefix character is used to specify type.

\$	(Dollar sign) specifies hexadecimal
@	(Commercial at) specifies octal
%	(Percent) specifies binary
'	(Apostrophe) specifies an ASCII literal character in immediate instructions

The absence of a prefix symbol indicates decimal value. In the statement

```
LDA    BETA+5
```

the decimal number 5 is added to BETA to compute the address. Similarly;

```
LDA BETA+$5F
```

denotes that the hexadecimal value 5F is to be added to BETA for the address computation.

The immediate mode of addressing is signified by a # (pounds sign) followed by a constant.

```
Example: LDA #2
```

specifies that the decimal value 2 is to be put into the accumulator. Similarly;

```
LDA      #'G
```

will load the ASCII character G into the accumulator.

Immediate mode addressing always generates two bytes of machine code, the OPCODE and the value to be used as OPERAND. Note that constant values can be used in address expressions and as values in immediate mode addressing. They can also be used to initialize locations as explained in a later section on assembler directives.

RELATIVE

There are 8 conditional branch instructions available to the user.

```
Example: BEQ      START      IF EQUAL BRANCH TO START
```

which might typically follow a compare instruction. If the values compared are equal a transfer to the instruction labeled START is made. The branch address is a one byte positive or negative offset which is added to the program counter during execution. At the time the addition is made the program counter is pointing to the next instruction beyond the branch instruction. Thus, a branch address must be within 127 bytes forward or 128 bytes backward from the conditional branch instruction. An error will be flagged at assembly time if a branch target falls outside the bounds for relative addressing. Relative addressing is not used for any other instructions.

IMPLIED

Twenty-five instructions such as TAX (Transfer Accumulator to Index X) require no OPERAND and hence are single byte instructions. Thus, the OPERAND addresses are

implied by the operation code.

Four instructions, ASL, LSR, ROL and ROR are special in that the accumulator, A, can be used as an OPERAND. In this special case, these four instructions are treated as implied mode addressing and only an operation code is generated.

INDEXED INDIRECT

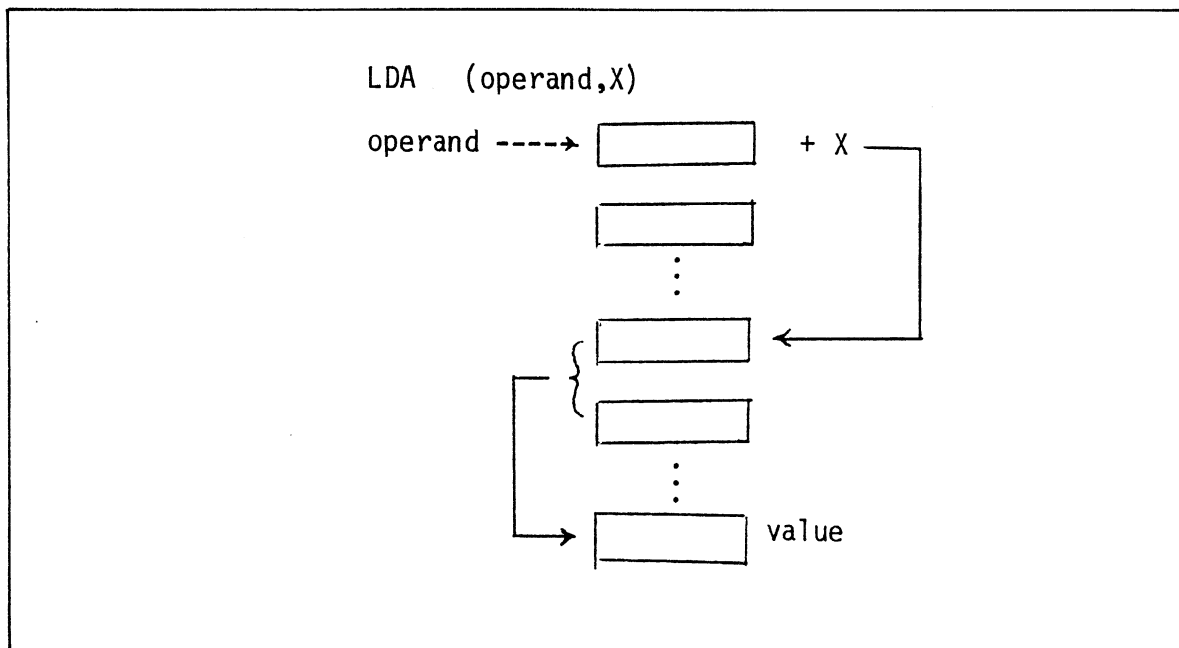
In this mode the OPERAND address is a location in page zero memory which contains the address to be used as an OPERAND.

Example: LDA (BETA,X)

The parentheses around the OPERAND indicates indirect mode. In the above example the value in index register X is added to BETA. That sum must reference a location in page zero memory. During execution the high order byte of the address is ignored, thus forcing a page zero address. The two bytes starting at that location in page zero memory are taken as the address of the OPERAND. For purposes of illustration assume the following:

BETA is 12
X contains 4
Locations 0017 and 0016 are 01 and 25
Location 0125 contains 37

Then BETA + X is 16, the address at location 16 is 0125. The value at 0125 is 37, and hence, the instruction LDA (BETA,X) loads the value 37 into the accumulator. This form of addressing is shown in the following illustration.



INDIRECT INDEXED

Another mode of indirect addressing uses index register Y and is illustrated by:

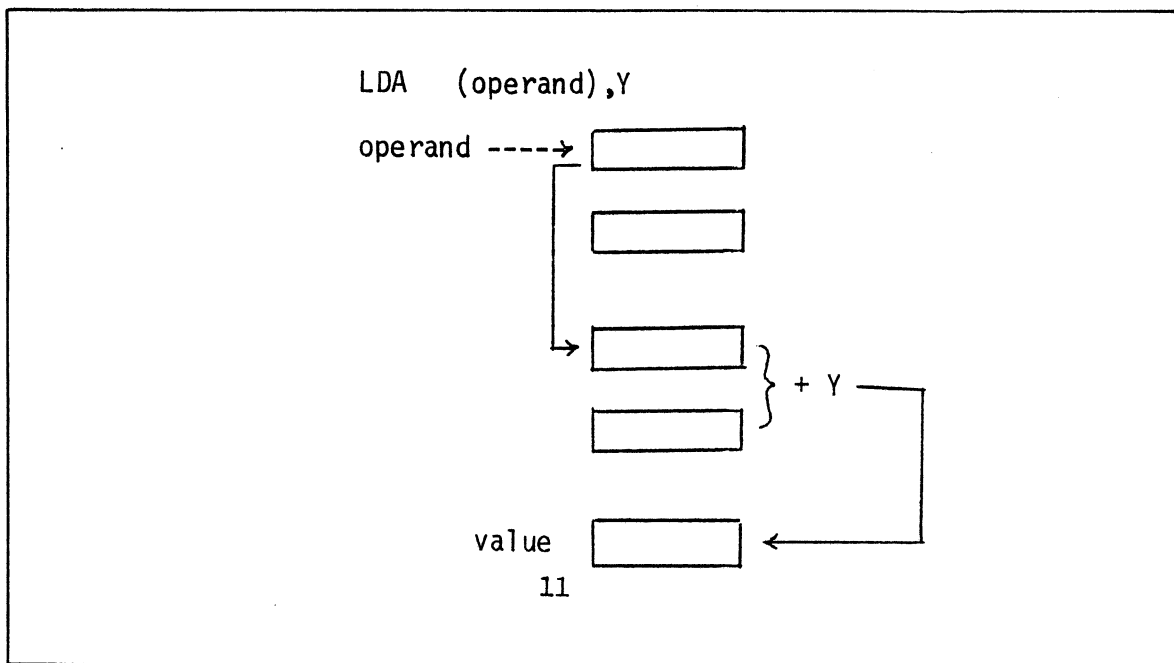
```
LDA (GAMMA),Y
```

In this case, GAMMA references a page zero location at which an address is to be found. The value in index Y is added to that address to compute the actual address of the OPERAND. Suppose for example that:

```
GAMMA is 38 (hexadecimal)
Y contains 7
Locations 0039 and 0038 are 00 and 54
Location 005B contains 126
```

Then the address at 38 is 0054 and 7 is added to this giving an effective address 005B. The value at 005B is 126 which is loaded into the accumulator.

In indexed indirect, the index X is added to the OPERAND prior to the indirection. In indirect indexed, the indirection is done and then the index Y is added to compute the effective address. Indirect mode is always indexed except for a JMP instruction which allows an absolute indirect address as exemplified by JMP (DELTA) which causes a branch to the address contained in locations DELTA and DELTA+1. The indexed indirect mode of addressing is shown in the following illustration.



Chapter 3

ASSEMBLER DIRECTIVES

There are ten directives used to control the assembly process, define values or initialize memory locations. Assembler directives always appear in the OPCODE field of an instruction and thus might be considered as assembly time OPCODES instead of execution time OPCODES. The directives are:

```
.BYTE    .WORD    .LIB    .DBYTE    .OPT
.PAGE    .SKIP    .FILE    * =    or    =    .END
```

All directives which are preceded by the period may be abbreviated to the period and three characters if desired e.g., .BYTE.

.BYTE is used to reserve one byte of memory and load it with a value. The directive may contain multiple OPERANDS which will store values in consecutive bytes. ASCII strings may also be generated by enclosing the string with quotes.

```
HERE      .BYTE 2
THERE     .BYTE 1, $F, @3, %101, 7
ASCII     .BYTE 'ABCDEFH'
```

Note that numbers may be represented in the most convenient form. In general, any valid MCS650X expression which can be resolved to eight bits may be used in this directive. If it is desired to include a quote in an ASCII string, this may be done by inserting two quotes in the string;

```
.BYTE 'JIM'S CYCLE'
```

could be used to print:

```
JIM'S CYCLE
```

.WORD is used to reserve and load two bytes of data at a time. Any valid expression, except for ASCII strings, may be used in the OPERAND field.

```
HERE      .WORD 2
THERE     .WORD 1, $FF03, @3
WHERE     .WORD HERE, THERE
```

The most common use for .WORD is to generate addresses as shown in the above example labelled "WHERE" which stores the 16 bit addresses of "HERE" and "THERE". Addresses in the MCS-650X are fetched from memory in the order low-byte and high-byte, therefore, .WORD generates the values in this order.

The hexadecimal portion of the example (\$FF03) would be stored 03,FF. If this order is not desired, use .DBYTE rather than .WORD. .DBYTE is exactly like .WORD except the bytes are stored in high-byte, low-byte order.

```
.DBYTE    $FF03
```

will generate FF,03. Thus, fields generated by .DBYTE may not be used as indirect addresses.

Equal (=) is the EQUATE directive and is used to reserve memory locations, reset the program counter (*), or assign a value to a symbol.

```
HERE      *=*+1      reserve one byte
WHERE     *=*+2      reserve two bytes
*=$200    set program counter
NB=8      assign value
MB=NB+%101 assign value
```

The = directive is very powerful and can be used for a wide variety of purposes.

Expressions must not contain forward references or they will be flagged as an error.

Example: * = C + D - E + F

would be legal if C, D, E and F are all defined, but would be illegal if any of the variables were a forward reference. Note also that expressions are evaluated in strict left to right order.

.PAGE is used to cause an immediate jump to top of page and may also be used to generate or reset the title printed at top of page.

```
.PAGE      'THIS IS A TITLE'
.PAGE
.PAGE      'NEW TITLE'
```

If a title is defined, it will be printed at the top of each page until it is redefined or cleared. A title may be cleared with:

```
.PAGE      '    '.
```

.SKIP is used to generate blank lines in a listing. The directive will not appear but its position may be found in a listing, since it is treated as a valid input "card" and the card number printed on the left side of the listing will jump by two when the next line is printed.

```
.SKIP      2          skip two blank lines
```

.SKIP	3*2-1	skip five lines
.SKIP		skip one line

.OPT is the most powerful directive and is used to control the generation of output fields, listings and expansion of ASCII strings in .BYTE directives.

.OPT	ERRORS, LIST, MEMORY
.OPT	NOE, NOL, NOM, NOG

Also valid is:

.OPT	ERR
------	-----

Default settings are:

.OPT	MEM, LIST, ERR, NOGEN
------	-----------------------

Here are descriptions for each of the OPERANDS:

ERRORS [NO ERRORS]:

Used to control creation of a separate error file. The error file contains the source line in error and the error message. This facility is normally of greatest use to time-sharing users who have limited print capacity. The error file may be turned on and examined until all errors have been corrected. This listing file may then be examined. Another possibility is to run with:

.OPT	ERRORS, NOLISTING
------	-------------------

until all errors have been corrected, and then make one more run with:

.OPT	NOERRORS, LISTING
------	-------------------

LIST [NOLIST]:

Used to control the generation of the listing file which contains source input, errors and warnings, code generation, symbol table and instruction count if enabled.

MEMORY [NOMEMORY]:

Used to control generation of the memory file, which is used as an interface between the assembler and the simulator and various loader programs. The memory file contains information about symbols, line numbers and code generation, and is described in detail elsewhere in this document.

GENERATE [NOGENERATE]:

Used to control printing of ASCII strings in the .BYTE directive. The first two characters will always be printed, and subsequent characters will be printed (normally two bytes per line), if GENERATE is used.

.END should be the last statement in a file and is used to signal the physical end of the file. Its use is optional, but highly recommended for program documentation.

.LIB directive allows the user to construct control files containing this directive which inserts the module named into the assembly. When the assembler encounters this directive, it temporarily ceases reading source code from the file containing it and starts reading from the file named on the .LIB. Processing of the original source file resumes when end-of-file (EOF) or .END is encountered in the library file. The control file containing the .LIB can contain other assembler directives to turn the listing function on and off etc..

A library file called by a .LIB may not contain another .LIB, but it may contain a .FIL. .FIL terminates assembly of the file containing it and transfers source reading to the file named in the OPERAND. There are no restrictions on the number of files which may be linked by .FIL directives. Caution should be exercised when using this directive to ensure that no circular linkages are created. An assembler pass can only be terminated by (EOF) or .END directive.

NOTES

Chapter

4

OUTPUT FILES

There are three output files generated by the assembler. Each file is optional through the use of the .OPT assembler directive. The listing file contains the program list, and symbol table. The error file contains all error lines and errors. The error file is included in the listing file. The interface file contains the object code for the loader.

LISTING FILE

The listing file will be produced unless the NOLIST option is used on the .OPT assembler directive. This file is made up of two sections: Program, and Symbol Table.

Program

This listing will always be produced unless the NOLIST option is selected. It contains the source statements of the program along with the assembled code. Errors and warnings appear after erroneous statements. An explanation of error codes are presented in part B. A count of the errors and warnings found during the assembly are presented at the end of the program.

Symbol Table

The symbol table will always be produced unless the NOSYM option is used. It contains a list of all symbols used in the program, and their addresses.

INTERFACE FILE

The format for the first and all succeeding records, except for the last record, is as follows:

```
; nln0 a3a2a1a0 (d1d0)1 (d1d0)2 x3x2x1x0
```

Where the following statements apply:

1. All characters (n,a,d,x) are the ASCII characters 0 through F, each representing a hexadecimal digit.
2. ; is a record mark indicating the start of a record.
3. nln0 = the number of bytes of data in this record

(in hexadecimal). Each pair of hexadecimal characters (dld0) represents a single byte in the record.

4. a3a2ala0 = the hexadecimal starting address for the record. a3 represents address bits 15 thru 12, etc.. The 8-bit byte represented by (dld0)1 is stored in address a3a2ala0; (dld0)2 is stored in (a3a2ala0) + 1, etc.
5. (dld0) = two hexadecimal digits representing an 8-bit byte of data. (dl = high-order 4 binary bits and d0 = low-order 4-bits). maximum of 18 (Hex) or 24 (decimal) bytes of data per record is permitted.
6. x3x2xlx0 = record check sum. This is the hexadecimal sum of all characters in the record, including the nln0 and a3a2ala0, but excluding the record mark and the check sum of characters. To generate the check sum, each byte of data (represented by two ASCII characters) is treated as 8 binary bits. The binary sum of these 8-bit bytes is truncated to 16 binary bits (4 hexadecimal digits) and is then represented in the record as four ASCII characters (x3x2xlx0).

The format for the last record in a file is as follows:

 ; 00 c3c2clc0 x3x2xlx0

1. ; 00 = zero bytes of data in this record. This identifies this as the final record in a file.
2. c3c2clc0 = the total number of records (in hexadecimal) in this file, NOT including the last record.
3. x3x2xlx0 = check sum for this record.

ERROR LIST

Error messages are given in the program listing accompanying statement in error . The following is a list of all error messages which might be produced during assembly.

****A,X,Y,S,P RESERVED**

A label on a statement is one of the five reserved names (A, X, Y, S and P). They have special meaning to the assembler and therefore cannot be used as labels. Use of one of these names will cause this error message to be printed and zero bytes to be generated for the statement. The label does not get defined and will appear in the symbol table as an undefined variable. Reference to such a label elsewhere in the program will cause error messages to be printed as if the label were never declared.

HOW TO AVOID: Don't use A, X, Y, S or P as a label to a statement.

****A MODE NOT ALLOWED**

Following a legal OPCODE, and one or more spaces, is the letter A followed by one or more spaces. The assembler is trying to use the accumulator (A = accumulator mode) as the OPERAND. However, the OPCODE in the statement is one which does not allow reference to the accumulator. Check for a statement labelled A (an illegal statement), which this statement is referencing. If you were trying to reference the accumulator, look up the valid OPERANDS for the OPCODE used.

****INVALID ADDRESS**

An address referred to in an instruction, or in one of the assembler directives (.BYTE, .DBYTE, .WORD), is invalid. In the case of an instruction, the OPERAND that is generated by the assembler must be greater than or equal to zero, and less than or equal to \$FFFF (2 bytes long). (This excludes relative branches which are limited to +/- 127 from the next instruction.) If the OPERAND generates more than two bytes of code or is less than zero, this error message will be printed. For a .BYTE each OPERAND is limited to one byte, and for a .WORD each OPERAND is limited to two bytes. All must be greater than or equal to zero.

This validity is checked after the OPERAND is evaluated. Check for values of symbols used in the OPERAND field (see the symbol table for this information).

****FORWARD REFERENCE**

The expression on the right side of an equals sign contains a symbol that hasn't been defined previously. One of the operations of the assembler is to evaluate expressions or

labels, and assign addresses or values to them. The assembler processes the input values sequentially, which means that all of the symbolic values that are encountered fall into two classes--already defined values and not previously encountered values. The assembler assigns defined values and builds a table of undefined values. When a previously used value is discovered, it is substituted into the table and the assembler processes all of the input statements a second time using currently defined values.

A label or expression which uses a yet undefined value is considered to be referenced forward to the to-be-defined value.

To allow for conformity of evaluating expressions, this assembler allows for one level of forward reference so that the following code is allowed.

Card Sequence	Label	Opcode	Operand
100		BNE	New One
200	New One	LDA	#5

But the following are not allowed:

Card Sequence	Label	Opcode	Operand
100		BNE	New One
200	New One		Next + 5
300	Next	LDA	# 5

This feature should not disturb the normal use of labels as the cure for this error.

Card Sequence	Label	Opcode	Operand
100		BNE	New One
300	Next	LDA	# 5
301	New One		Next + 5

is very simple and always solves the problem.

This error may also mean that the value on the right side of the = is not defined at all in the program; in which case, the cure is the same as for undefined values.

The assembler cannot process more than one computed forward reference. All expressions with symbols that appear on the right side of any equal sign must refer only to previously defined symbols for the equate to be processed.

****ILLEGAL OPERAND TYPE**

After finding an OPCODE that does not have an implied OPERAND, the assembler passes the OPERAND field (the next non-blank field following the OPCODE) and determines what

type of OPERAND it is (indexed, absolute,, etc.). If the type of OPERAND found is not valid for the OPCODE, this error message will be printed.

Check to see what types of OPERANDS are allowed for the OPCODE and make sure the form of the OPERAND type is correct (see the section on addressing modes).

Check for the OPERAND field starting with a left paren. If it is supposed to be an indirect OPERAND, recheck the correct format for the two types available. If the format was wrong (missing right paren or index register), this error will be printed. Also check for missing or wrong index registers in an indexed OPERAND (form: expression, index register).

****IMPROPER OPCODE**

The assembler searches a line until it finds the first non-blank character string. If this string is not one of the 56 valid OPCODES it assumes it is a label and places it in the symbol table. It then continues parsing for the next non-blank character string. If none are found, the next line will be read in and the assembly will continue. However, if a 2nd field is found it is assumed to be an OPCODE (since only one label is allowed per line). If this character string is not a valid OPCODE, the error message is printed.

This error can occur if OPCODES are misspelled, in which case the assembler will interpret the OPCODE as a label (if no label appears on the card). It will then try to assemble the next field as the OPCODE. If there is another field, this error will be printed.

Check for a misspelled OPCODE or for more than one label on a line.

****CAN'T EVAL EXPRESSION**

In evaluating an expression, the assembler found a character it couldn't interpret as being part of a valid expression. This can happen if the field following an OPCODE contains special characters not valid within expressions (e.g. parentheses). Check the OPERAND field and make sure only valid special characters are within a field (between commas).

****INDEX MUST BE X OR Y**

After finding a valid OPCODE, the assembler looks for the OPERAND. In this case, the first character in the OPERAND field is a left paren. The assembler interprets the next field as an indirect address which, with the exception

of the jump statement, must be indexed by one of the index registers, X or Y. In the erroneous case, the character that the assembler was trying to interpret as an index register was not X or Y and the error was printed.

Check for the OPERAND field starting with a left paren. If it is supposed to be an indirect OPERAND, recheck the correct format for the two types available. If the format was wrong (missing right paren or index register), this error will be printed. Also check for missing or wrong index registers in an indexed OPERAND (form: expression, index register).

****LABEL START NEED A-Z**

The first non-blank field is not a valid OPCODE. Therefore, the assembler tried to interpret it as a label. However, the first character of the field does not begin with an alphabetic character and the error message is printed.

Check for an unlabelled statement with only an OPERAND field that does start with a special character. Also check for illegal label instruction.

****LABEL TOO LONG**

All symbols are limited to six characters in length. When parsing, the assembler looks for one of the separating characters to find the end of a label or string. If other than one of these separators is used, the error message will be printed providing that the illegal separator causes the symbol to extend beyond six characters in length. Check for no spacing between labels and OPCODES. Also check for a comment card with a long first word that doesn't begin with a semicolon. In this case the assembler is trying to interpret part of the comment as a label.

****NON-ALPHANUMERIC**

Labels are made up of from one to six alphanumeric digits. The label field must be separated from the OPCODE field by one or more blanks. If a special character or other separator is between the label and the OPCODE, this error message might be printed.

The 56 valid OPCODES are each three alphabetic characters. They must be separated from the OPERAND field (if one is necessary) by one or more blanks. If the OPCODE ends with a special character (such as a comma), this error message will be printed.

In the case of a lone label or an OPCODE that needs no OPERAND, they can be followed directly by a semicolon to

denote the rest of the card as a comment (use of a semicolon tabs the comment out to the next tab position).

****DUPLICATE SYMBOL**

The first field on the card is not an OPCODE so it is interpreted as a label. If the current line is the first line in which that symbol appears as a label (or on the left side of an equals sign), it is put in the symbol table and tagged as defined in that line. However, if the symbol has appeared as a label, or on the left of an equate, prior to the current line, the assembler finds the label already in the symbol table. The assembler does not allow redefinitions of symbols and will, in this case, print the error message.

****INDIRECT OUT OF RANGE**

An indirect address is recognized by the assembler by the parentheses that surround it. If the field following an OPCODE has parens around it, the assembler will try to assemble it as an indirect address. If the OPERAND field extends into absolute (is larger than 255 - one byte), this error message will be printed.

This error will only occur if the OPERAND field is in correct form (i.e. an index register following the address), and the address field is out of page zero. To correct this, the address field must refer to page zero memory.

****PC NEGATIVE — RESET 0**

An assembled program is loaded into core from position 0 to 64K (65535). This is the extent of the machine. Instructions can only refer to up to 2 bytes of information. Because there is not such a thing as negative memory, an attempt to reference a negative position will cause this error and the program counter (or pointer to the current memory location), will be reset to 0.

When this error occurs, the assembler continues assembling the code with the new value of the program counter. This could cause multiple bytes to be assembled into the same locations. Therefore, care should be taken to keep the program counter within the proper limits.

****RAN OFF END OF CARD**

This error message will occur if the assembler is looking for a needed field and runs off the end of the card (or line image) before the field is found. The following

should be checked for: a valid OPCODE field without an OPERAND field on the same card: an OPCODE that was thought to take an implied OPERAND, which in fact needed an OPERAND: an ASCII string that is missing the closing quote (make sure any embedded quotes are doubled - to have a quote in the string at the end, there must be 3 quotes - 2 for the embedded quote and one to close off the string); a comma at the end of the OPERAND field indicates there are more OPERANDS to come: if there aren't other OPERANDS, the assembler will run off the card looking for them.

****BRANCH OUT OF RANGE**

All of the branch instructions (excluding the two jumps), are assembled into 2 bytes of code. One byte is for the OPCODE and the other for the address to branch to. To allow a forward or backward branch, this branch is taken relative to the beginning of the next instruction, according to the address byte. If the value of the byte is 0-127 the branch is forward; if the value is 128-255 the branch is backward. (A negative branch is in 2's complement form). Therefore, a branch instruction can only branch forward or backward 127 bytes relative to the beginning of the next instruction. If an attempt is made to branch further than these limits, the error message will be printed.

****UNDEFINED DIRECTIVE**

All assembler directives begin with a period. If a period is the first character in a non-blank field the assembler interprets the following character string as a directive. If the character string that follows is not a valid assembler directive, this error message will be printed.

Check for a misspelled directive, or a period at the beginning of a field that is not a directive.

****UNDEFINED SYMBOL**

This error is generated by the second pass. If, in the first pass the assembler finds a symbol in the OPERAND field (the field following the OPCODE or an equals sign), that has not been defined yet, that symbol is flagged for interpretation by pass two. If the symbol is defined (shows up on the left of an equate or as the first non-blank field in a statement), pass one will define it and enter it in the symbol table. Then a symbol in an OPERAND field before the definition will be defined with a value when pass two assembles it. In this case the assembly process can be

completed.

However, if pass one doesn't find the symbol as a label or on the left of an equate, it never enters it in the symbol table as a defined symbol. When pass two tries to interpret the OPERAND field this type of symbol is in, there is no value for the symbol and the field cannot be interpreted. Therefore, the error message is printed with no value for the OPERAND.

This error will also occur if a saved symbol A, X, Y, S or P, is used as a label and referred to elsewhere in the program. On the statement that references the saved symbol, the assembler sees it as a symbol that has not been defined.

Check for use of saved symbols, misspelled labels or missing labels to correct this error.

NOTE:

When the assembler finds an expression (whether it is in an OPERAND field or on the right of an equals sign) it tries to evaluate the expression. If there is a symbol within the expression that hasn't been defined yet, the assembler will flag it as a forward reference and wait to evaluate it in the second pass. If the expression is on the right side of an equal sign, the forward reference is a severe error and will be flagged as such. However, if the expression is in an OPERAND field of a valid OPCODE, the first pass will set aside 2 bytes for the value of the expression and flag it as a forward reference. When the 2nd pass fills in the value of the expression, and the value of the expression is one byte long i.e., <256, the instruction is one byte longer than required, because the forward reference to page zero memory wastes one byte of memory - the extra one that was saved, because, during the first pass, the assembler didn't know how large the value was, so had to save for the largest value - two bytes.

NOTES

UNIVERSAL DOS SUPPORT

This program is an aid to the 2040 user. When the program is operating, the user may enter disk commands and interrogate the 2040's error channel without using BASIC commands.

When the wedge program is running it is 'wedged' into the operating system and BASIC interpreter. Thus the program can trap keyboard input before BASIC sees it. This is done by linking into the CHRGET routine in page zero.

The operation of the wedge program is accomplished using three commands that are typed in the first character position of a line. The greater-than symbol (>) is used to print the directories, send commands and read the error channel. The following examples illustrate the usage of the > command:

>I0	Initialize drive 0
>S1:S*	Scratch all files on drive one that start with the letter S
>\$1	Read the directory on drive 1 and print it to the PET screen
>\$0:BASIC*	Read the directory on drive 0 and search for filenames starting with the string 'BASIC'

The most notable attribute about this method of reading the directory is that it does not destroy programs in memory so you may examine the directory at any time.

The third use of the > command is interrogation of 2040 error channel. This is done by typing:

>(RETURN)

the computer responds by printing the error message on the screen. This does the same thing as the following BASIC program:

```
10 OPEN15,8,15
20 INPUT#15,A,B$,C,D
30 PRINTA;B$;C;D
```

NOTE: After a cold start, do not attempt to read the error channel before sending a command.

Business keyboard users may substitute the @ (commercial at) symbol in place of >.

The second command is the slash command. It performs the same function as the BASIC 'LOAD' with a simplified format:

`/ASSEMBLER`

The above command loads the program named 'ASSEMBLER'. This does the same thing as the following BASIC command.

```
LOAD"ASSEMBLER",8
```

The third program is the LOAD/RUN command. This command is implemented using the up-arrow key .

HURKLE

This example would load then run the program HURKLE.

The wedge program is the first program on the development disk, thus it may be loaded with the `LOAD"*,8` . This command causes an automatic initialization on Drive 0 when executed as the first disk command after a cold start.

NOTES

Appendix II

INSTRUCTIONS FOR THE MINI-EDITOR

The MINI-EDITOR is included on the diskette supplied with this manual. There are two versions; 16K-EDITOR and 32K-EDITOR, used in 16K and 32K machines respectively. The only difference between the two versions is the load point.

The editor is used to enter and modify source files for the assembler. The editor retains all of the features of the screen editor plus; automatic line numbering, find, change, delete with range, and renumber. Other commands include get, put, break, kill, and format. All of the commands are detailed in the summary at the end of this appendix.

This is a line number oriented editor but with the functions of the screen editor it can be considered to have a character mode which includes the lines appearing on the screen. The editor commands operate in a similar fashion to the commands already existing in the computer's BASIC. There are several example files included on the diskette with the editor. Users would be well advised to try out the editor on these files in order to familiarize themselves with the commands.

The data files on which the assembler operates are ASCII characters with each line terminated by a carriage return. The only restriction on data files is in naming. Due to the method in which the assembler parses, you are not allowed spaces in filenames. The files are sequential and must be terminated by a zero byte \$00.

LOADING THE MINI-EDITOR

The editor can be loaded with the wedge load command or with a BASIC load command:

```
/32K-EDITOR    or  
LOAD"16K-EDITOR",8
```

To initiate the editor use the SYS command. The editor is started with the following commands, SYS7*4096 for the 32K version, and SYS3*4096 for the 16K editor. After typing the SYS command the editor will respond with the message 'MINI-EDITOR V111679'. At this point type a NEW command to clear the text pointers. You are now ready to edit assembler source files.

OPERATION OF THE MINI-EDITOR

When the MINI-EDITOR is in operation any statement typed:

```
10 FOR I=1 TO 10
```

will not be tokenized. Thus, you cannot type a BASIC line with the editor turned on. To avoid the above problem you must disable the editor with the 'KILL' command or reset to start clean.

Source files are loaded with the 'GET' command. As the file is loaded the editor adds line numbers. The editor starts its numbering at 1000. After editing your file, insure that the last line is a .FILE or a .END assembler directive. Then save your file on disk with the 'PUT' command.

The repeat key is enabled by typing (return). All of the keys on the keyboard will repeat when held down for more than one-half of a second. The repeat function is still operational after the editor has been killed. To disable the repeat function type: SHIFT-RUN/STOP followed by RUN/STOP (for version 2.0 BASIC). Type LOAD(return) followed by RUN/STOP (for version 4.0).

MINI-EDITOR COMMANDS

Auto Line Numbering

The function of this command is to generate new line numbers for entry of source code. In order to enable the auto command type the following:

```
AUTO nl(return)
```

where nl is the increment between line numbers printed. To disable the auto function type the auto command without an increment.

Break Command

The BREAK command jumps to the ROM resident monitor. This command is executed by typing:

```
BREAK(return)
```

Change String

The change command allows the user to automatically

locate and replace one string with another (multiple occurrences). This command is entered in the following format:

CHANGE/str1/str2/,nl-n2,Q

/	Delimits the str1 and str2 (any character not in either str)
str1	Search string
str2	Replacement string
,nl-n2	Range paramaters. The format is the same as the BASIC LIST command. If omitted the whole file is searched.
,Q	Query parm. if this parm is included the editor will prompt the user for a yes or no response to the change

Delete

A delete range function has been included to make deletion quicker. The format is the same as the BASIC list command:

DELETE nl-n2

Ensure that you use the range parms, as leaving them out causes the entire file to be deleted.

Find String

The find command is used to locate specific strings in text. Each occurrence of the string is printed on the CRT. You can halt the printing with the RUN/STOP key. Printing can then be continued with the space bar or terminated with the equal key. The format of the FIND command is:

FIND/str1/,nl-n2

/	Delimiter
str1	Search string
,nl-n2	Range parm. same as BASIC list command

Formatted Print

The format command is used to print the text file in tabbed format like the assembler. For this function to work correctly you must type mnemonics in column two, or one space from labels.

```
FORMAT    n1-n2
           n1-n2      Range parms of the same
                       format as LIST.
```

Note: This command has the same controls as find. For example, RUN/STOP halts printing, SPACE restarts, and equal '=' quit.

GET Files

Input assembler text files from disk. This command is used to load source files into the editor and append to files already in memory.

```
GET "FILE-NAME",n1,n2,n3
           n1          Begins inputting source
                       at this line.
           n2          Device #, default is 8
           n3          Secondary address
                       default is 8
```

Note: GET starts numbering at 1000 by 10.

KILL Command

This command causes the editor to disengage. This does not disable the repeat function if it has been turned on prior to KILL. To restart the editor, type the same command used to start the editor.

Resequenece Lines

The NUMBER function allows the user to renumber all or part of the file in memory.

```
NUMBER n1,n2,n3
           n1          Old start line number
           n2          New start line number
           n3          Step size for resequence
```

PUT Command

The PUT command outputs source files to the floppy for later assembly. PUT has the ability to output all or part of the memory resident file.

```
PUT "0:NAME",n1-n2,n3,n4
      0:      Drive number, for disk only
      NAME    Output file name
      n1      Starting line number
      n2      Ending   line number
      n3      Device #, default is 8
      n4      Secondary address,
              default is 8
```

If n1-n2,n3,n4 are left out, the whole file is outputted to the default device.

MINI-EDITOR COMMAND SUMMARY

COMMAND	DESCRIPTION
AUTO n1	Starts automatic line numbering.
AUTO	Shuts off auto
BREAK	Jump to the monitor
CHANGE/s1/s2/,n1-n2	Change string
CHANGE/s1/s2/	Change string no range
DELETE n1-n2	Delete range
FIND/s1/,n1-n2	Find string
FIND/s1/	Find string no range
FORMAT n1-n2	Print formatted
GET"FILE",n1-n2,n3	Bring in text
GET"FILE"	Short form get
KILL	Disable the editor
LIST	List lines of text
NUMBER n1,n2,n3	Renumber text
PUT"0:FILE",n1-n2,n3,n4	Save text on disk
PUT"1:FILE"	Save text short form

NOTES

Appendix

OPERATION OF THE PET ASSEMBLER

The assembler loads as a BASIC program. To start the assembler, enter a RUN command. This works because the program starts with a SYS command which is set up in the assembler source. An example of how to start a program in this manner is given just before the appendices of this manual. When the assembler is run it will print a copyright notice and print the first user prompt.

The first user prompt is for an object filename. If no object file is desired, type a return for this prompt. If an object file is desired the response would be something like:

```
OBJECT FILE? 1:TESTOBJ
```

Note that a drive number is specified before the filename.

The second prompt is for hard copy. A null response will produce output. You must type N for no, followed by a return to defeat the printed listing.

The hard copy may be directed to the IEEE port (device #4) or to the User Port. (User Port handshake is Centronics parallel; see the notes at the end of this appendix for details.). The user determines which method by responding Y or N to the IEEE Printer question. A null response defaults to a yes.

The final prompt is for the source filename. When the user types the name of the file to be assembled followed by return, the assembler starts operation. If a null response is detected the assembler returns control to BASIC. This allows the user to correct mistakes.

Upon startup the assembler initializes both of the 2040 disk drives, opens the source file and starts the assembly. If an object file has been requested it is also opened.

There are several errors which may occur on a system level rather than an assembly level. These errors are caused by disk problems and user errors. They are generally easy to solve as is presented in the following examples.

The first is 'FILE NOT FOUND'; which is produced when one of the following occurs:

1. The source file was not found.
2. A .LIB specifies a nonexistent file.
3. A .FIL specifies a nonexistent file.

This error is of the human type: e.g., the user has mistyped a filename or placed the wrong diskette into the floppy.

The second error is 'FILE EXISTS'. This error is produced when the object file named already exists on the drive specified. This error can be cured by scratching the old file or changing the diskette.

The third error is 'READ ERROR'. This error is a disk read error. Please refer to the 2040 user manual for a description of the errors and their causes.

Notes on Operation

When the assembler is running, operation may be halted by pressing the RUN/STOP key. This only halts the assembly process; operation may be terminated at this point by pressing the B or T key. These keys return control to BASIC or TIM respectively. Pressing any other key continues the assembly process. This feature is useful for users without printers, as the screen listing can be examined during assembly.

The assembler can send the printed output to the User Port with a Centronics parallel handshake method. This allows the user to attach a printer other than a Commodore printer. The data is transferred byte parallel with two handshake lines, Data Strobe and Acknowledge (both active low). The data is placed on the port then the Data Strobe is pulled low 6 usecs later, starting the handshake sequence. The computer now waits for the acknowledge line to be pulled low by the printer. When this happens, the data strobe line is released and the handshake sequence is completed for one character. The following presents the interconnection for the interface. The user should keep cable lengths to a minimum, as there are no line drivers in the computer on the output lines. The user should also check the loading of the device that is being connected because the computer can only drive one standard TTL load.

Interconnection Table

PET	Description	Centronics
M	(cb2) Data Strobe	1
C	(pa0) Data 1	2
D	(pa1) Data 2	3
E	(pa2) Data 3	4
F	(pa3) Data 4	5
H	(pa4) Data 5	6
J	(pa5) Data 6	7
K	(pa6) Data 7	8
L	(pa7) Data 8	9
B	(cal) Acknowledge	10
N	(gnd) Ground	16
A	(gnd) Ground	33

Appendix IV

PET LOADERS

The PET Assembler produces portable output in an ASCII format that is not directly executable. This output must be LOADED so the program can be executed. This is the function of a Loader.

There are three versions of the Loader included on the development disk. Each version is positioned in a different area of RAM memory. This allows the user to load anywhere in RAM by using the correct loader. The following table shows the names, load points and run commands for each of the three loaders.

NAME	LOAD ADDRESS	RUN COMMAND
LOADER	\$0400	RUN
MID-LOADER	\$3000	SYS 3 *4096
HI-LOADER	\$7000	SYS 7 *4096

The loaders are about 512 bytes long and all operate in the same manner. When activated, the loaders print a copyright notice and prompt the user for a load offset. The offset is used to place object code into an address range other than the one that it was assembled into. This allows the user to assemble for an area where there is no RAM and load into a RAM area. The object can then be programmed into EPROM etc..

The offset is a two byte hexadecimal address that is added to the program addresses. If the program address, plus the offset, is greater than \$FFFF, the address wraps around through \$0000. The following examples show how offset works.

Address	Offset	Load Point
\$0400	\$0000	\$0400
\$3000	\$0000	\$3000
\$9000	\$D000	\$2000
\$E000	\$4000	\$2000

After the offset is entered, the loader will prompt the user for the object filename to be loaded. The loader will then initialize both drives, search for the file and start the load. As the data is loaded, the program will print the input data to the CRT. This is for user feedback only. When the load is completed the loader prints the message 'END OF LOAD' and returns to BASIC.

There are three errors that can happen during a load. Errors are considered fatal; the load is terminated, the object file is closed, and control is returned to BASIC if an error is encountered. The following is a list of possible errors, which should be self documenting.

BAD RECORD COUNT
NON-RAM LOAD
CHECKSUM ERROR

CAUTION: The HI-LOADER (MID-LOADER for 16k machines) and the MINI-EDITOR load into the same RAM area. You must cold start the computer before using these loaders!

NOTES

Appendix V

I/O MAP FOR PET

CBM 2001 SERIES

6520

E810	DIAGNOSTIC SENSE	IEEE EOI INPUT	CASSETTE SENSE # 2 # 1	KEYBOARD ROW SELECT			PA	59408	
E811	TAPE #1 INPUT FLAG	• • •	SCREEN BLANK OUTPUT	CA2	DDRA ACCESS	CASSETTE #1 READ CONTROL	CA1	59409	
E812	KEYBOARD ROW INPUT							PB	59410
E813	RETRACE INTERRUPT FLAG	• • •	CASSETTE #1 MOTOR OUTPUT	CB2	DDRB ACCESS	RETRACE INTERRUPT CONTROL	CB1	59411	

6520

E820	IEEE INPUT						PA	59424
E821	ATN INTERRUPT FLAG	• • •	IEEE NDAC OUTPUT	CA2	DDRA ACCESS	IEEE ATN IN CONTROL	CA1	59425
E822	IEEE OUTPUT						PB	59426
E823	SRQ INTERRUPT FLAG	• • •	IEEE DAV OUTPUT	CB2	DDRB ACCESS	IEEE SRQ INPUT	CA1	59427

6522

E840	DAV INPUT	NRFD INPUT	RETRACE INPUT	CASSETTE #2 MOTOR OUTPUT	CASSETTE WRITE DATA (BOTH)	ATN OUTPUT	NRFD OUTPUT	NDAC INPUT	PB	59456
E841	PARALLEL USER PORT (P.U.P.) WITH HANDSHAKE								PA	59457
E842	DDRB (FOR \$E840)									59458
E843	DDRA (FOR \$E841, \$E84F)									59459
E844	TIMER 1								LOW	59460
E845									HIGH	59461
E846	TIMER 1 (LATCH)								LOW	59462
E847									HIGH	59463
E848	TIMER 2								LOW	59464
E849									HIGH	59465
E84A	SHIFT REGISTER									59466
E84B	TIMER 1 CONTROL PB7 OUTPUT	TIMER 1 CONTROL ONE-SHOT/FREE-RUN	TIMER 2 CONTROL	SHIFT REGISTER CONTROL			PB, PA LATCH CONTROL			59467
E84C	CB2 CONTROL (P.U.P. PIN #M)			CB1 INPUT	CA2 CONTROL (GRAPHICS/LOWER CASE)			CA1 INPUT		59468
E84D	IRQ STATUS	TIMER 1 INTERRUPT	TIMER 2 INTERRUPT	CB1 INTERRUPT	CB2 INTERRUPT	SHIFT REGISTER INTERRUPT	CA1 INTERRUPT	CA2 INTERRUPT		59469
E84E	ENABLE CLEAR/SET	T1 INTERRUPT ENABLE	T2 INTERRUPT ENABLE	CB1 INTERRUPT ENABLE	CB2 INTERRUPT ENABLE	SR INTERRUPT ENABLE	CA1 INTERRUPT ENABLE	CA2 INTERRUPT ENABLE		59470
E84F	USER PORT / NO HANDSHAKE								PA	59471
	7	6	5	4	3	2	1	0		



Commodore Business Machines, Inc.
3330 Scott Boulevard
Santa Clara, California 95051

**scanned by:
commodore
international
historical
society**

<https://www.commodore.international>